

# **The Policy Machine:**

## **A mechanism for the specification and enforcement of arbitrary attribute-based access control policies**

David Ferraiolo, Serban Gavrila, Steve Quirolgico  
National Institute of Standards and Technology

### **1 INTRODUCTION**

As a major component of any operating system or application, access control mechanisms come in a wide variety of forms, each with their individual methods for authentication, access control data constructs for specifying and managing policy, functions for making access control decisions and enforcement of policies, and a scope of protection that includes a defined set of users and resources.

#### **1.1 A lack of interoperability**

A natural consequence of the deployment of a multitude of heterogeneous mechanisms is a lack of interoperability. Although lack of interoperability may not be a problem for systems that can adequately operate independently of one another, access control mechanisms clearly do not fall into this category of systems. Access control policies are global and as such span systems and applications. Users with vastly different attributes and credentials have a need to access resources protected under different mechanisms, and resources that are protected under different mechanisms differ vastly in their sensitivity, and therefore accessibility. This lack of interoperability in today's access control paradigm introduces significant privilege and identity management challenges.

Lack of interoperation is but one problem with today's access control paradigm. Another pertains to policy enforcement.

#### **1.2 Limited policy enforcement**

Of the numerous recognized access control policies, today's OSs limit enforcement to instances of Discretionary Access Control (DAC) [1] and simple variations of Role-based Access Control (RBAC) [5, 9] policies, and to a far lesser extent, instances of Multi-level Security (MLS) [2] policies. Even within the enforcement of this narrow set of policies, issues exist. DAC and RBAC are considered weak in that that users (through overt actions and mistakes) and malicious code embedded within applications can potentially leak sensitive data to unauthorized users. Objects are also often under-protected under DAC and RBAC alone. For example, although access to medical records may be restricted to users in the role Doctor, not all doctors may have access to all medical records. Depending on the institution, other policies may come into play. Medical records

can be classified, only accessible to those doctors in a particular ward, or accessible only under the discretionary permission of a primary physician. This suggests a general need to protect objects under arbitrary combinations of policies. In their protection of classified information, MLS mechanisms can impose user and administrative inconveniences. At any moment in time users are prevented access to information for which they are otherwise legitimately authorized. And, MLS mechanisms are heavy handed in that in their narrow protection of classified information they require the labeling of, and include unclassified information in their scope of protection. In regard to many other recognized policies, there is a lack of any commercially viable OS mechanism for their enforcement.

To fill policy voids, policies are routinely accommodated through the implementation of access control mechanisms within applications. Prominent among these applications are database management systems, but these applications can also include a number of smaller applications such as enterprise calendars, and time and attendance. In many respects, these applications provide their services through the enforcement of access control policy. E-mail provides for the reading of messages and attachments through the discretionary distribution of objects, and workflow management provides for the reading and writing of specific documents to a prescribed sequence of users. Essentially, any application that requires a user's authentication, typically also affords access control services. Not only do these applications further aggravate identity and privilege management problems, but applications can also undermine policy enforcement objectives. For instance, although a file management system may narrowly restrict user access to a specific file, chances are the content of that file can be copied to an attachment or a message and mailed to anyone in the organization, or for that matter, the world.

### **1.3 The Policy Machine**

To solve the interoperability and policy enforcement problems of today's access control paradigm, NIST has developed an access control mechanism, referred to as the Policy Machine (PM). The PM is defined in terms of a fixed set of configurable data relations and a fixed set of functions that are generic to the specification and enforcement of combinations of a wide set of attribute-based access control policies. The PM is not an extension or adaptation of any existing access control model or mechanism, but instead is a redefinition of access control in terms of a fundamental and reusable set data abstractions and functions. Its objective is to provide a unifying framework to support not only currently enforceable policies, but also a host of orphan policies for which no mechanism yet exists for their viable enforcement. The PM requires changes only in its data configuration in the enforcement of arbitrary and organization-specific, attribute-based access control policies.

### **1.4 The Policy Machine**

We believe that the PM as currently specified and implemented represents a shift not only in the way we can specify and enforce policy, but also in the way we can effectively

develop and provide application services. We are now able to assert a number of benefits over the existing access control paradigm:

- **Policy flexibility** – Virtually any collection of attribute-based access control policies can be configured and enforced (e.g., DAC [1], MLS [2], Chinese wall [3], ORCON [7], object-based SoD constraints [10], etc.). In addition, basic application services can be provided through PM configuration to include those services offered by workflow management, email, and database management applications. This is in contrast to the “hard-wiring” of policy into the mechanism.
- **Policy combinations** – Resources (objects) regardless of their type can be selectively protected under one or more configurable policies (e.g., DAC only, or DAC and RBAC combined).
- **Comprehensive enforcement** – All user and subject (process) access requests, and all exchange of data to and from and among applications, between sessions, all exportation of data outside the bounds of the PM can be uniformly controlled under the protection policies of the objects of concern. (logical)
- **Assurance** – Configuration strategies can render malicious application code harmless, prevent unlawful leakage of data, and all enforcement could be implemented at the kernel level.
- **Policy libraries** – Standard configurations for a variety of policies are available and new configurations can be created for immediate policy instantiation, testing and deployment. This reduces the burden on administrators in specifying and configuring policies. In addition, basic application services can be provided through PM configuration to include those services offered by workflow management, email, and database management applications.

PM features as described above could be provided through a number of PM architectural deployments to include its implementation within a single operating system environment. Our reference implementation (embodiment of the PM), provides centralized policy configuration and decision-making within a local user environment. This kind of PM deployment affords still additional benefits, to include:

- **Single enterprise-wide scope of protection** – One administrative domain vs. policy management on an OS-by-OS and application-by-application basis. Access control policies are uniformly enforced over resources that are physically stored on a multitude of heterogeneous systems.
- **True single-sign on** – By virtue of the PM’s single scope of control, and a personal object system (POS) that includes the ability to reference and open any resource accessible to a user (e.g., email messages, work items, files, records and fields within records), the PM eliminates the need for a user to authenticate to a multitude of applications and hosts.

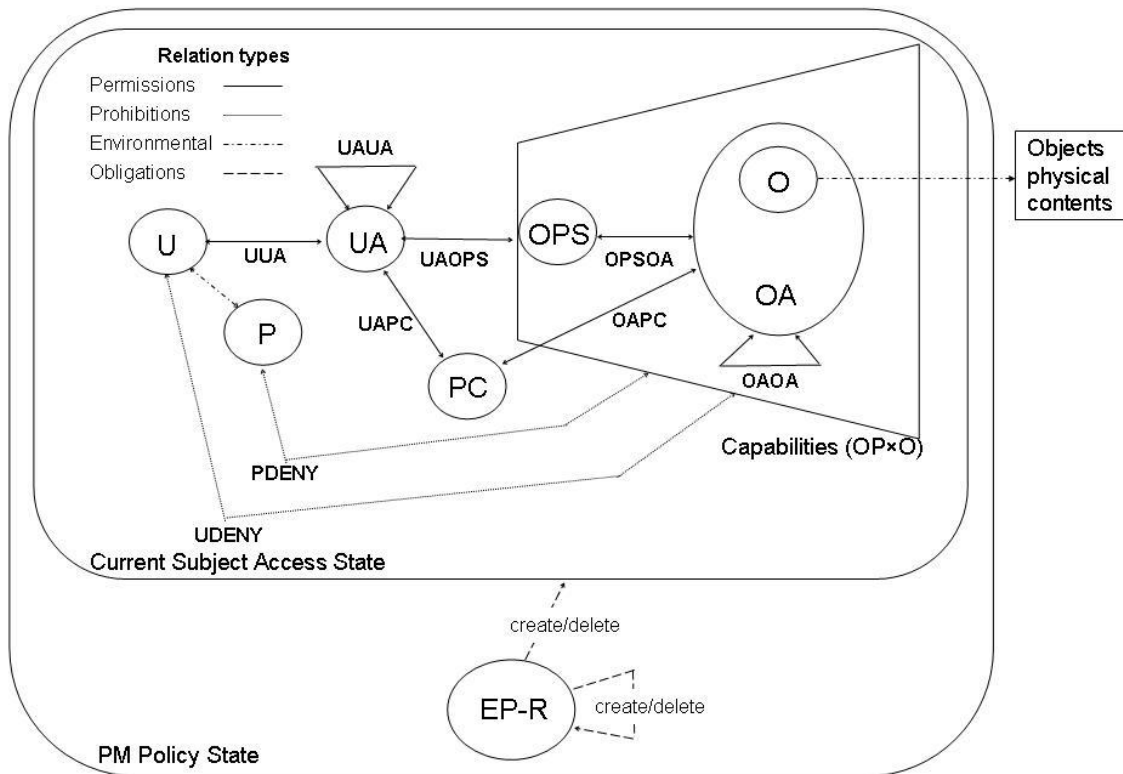
- **Logical access** – Any accessible resource could be securely accessed through any PM compliant OS with access to an application to process the resource.
- **Minimized OS Vendor support** – To be PM compliant, all an OS vendor needs to do is implement a standard set of enforcement functions (i.e., PM authentication, user resource presentation, session management and reference mediation), and does not need to be concerned with the management of access control data, or performing access control decisions.

With the PM's advantages over the existing access control paradigm, coupled with a minimum investment on the part of OS vendors, we see a strong business argument in favor of the adoption of the PM. Moreover, the features that provide these benefits are native to the PM and as such are afforded without the deployment and expense of less effective privilege management, provisioning, or identity management products.

The remainder of this document is organized as follows. Section 2 specifies the PM's data sets and relations under which a wide set of access control policies can be configured. Section 3 describes user authentication, user resource display, process creation, reference mediation, and inter-process communication functions that comprise the PM's operational environment. Section 4 provides a few illustrated examples of the PM's ability to configure and enforce policy. Section 5 is a description of an architectural PM deployment that can be applied to achieve cross platform enforcement of common access control policies. Section 6 is an illustration of the PM's ability to provide application services to include e-mail and workflow management through PM configuration.

## 2 PRIMITIVE SETS AND RELATIONS

PM relations, as depicted in Figure 1, are of three types – permissions, prohibitions, and obligations. Further depicted in Figure 1 is the decomposition of relations that define the PM policy. The configuration of permissions and prohibitions define the access state, and the access state and the obligation relations define the overall PM policy. The current access state defines the set of permissible user and subject (processes) accesses. Obligations dynamically alter PM relations to include current access relations and obligations themselves as a response to subjects accessing resources. While Figure 1 is a general illustration of PM relations, Figure 2 depicts an example configuration of PM relations in expressing policy.



**Figure 1. PM policy state data and relations**

Prior to defining PM relations we introduce the PM's basic and aggregated sets.

## 2.1 Primitive data sets

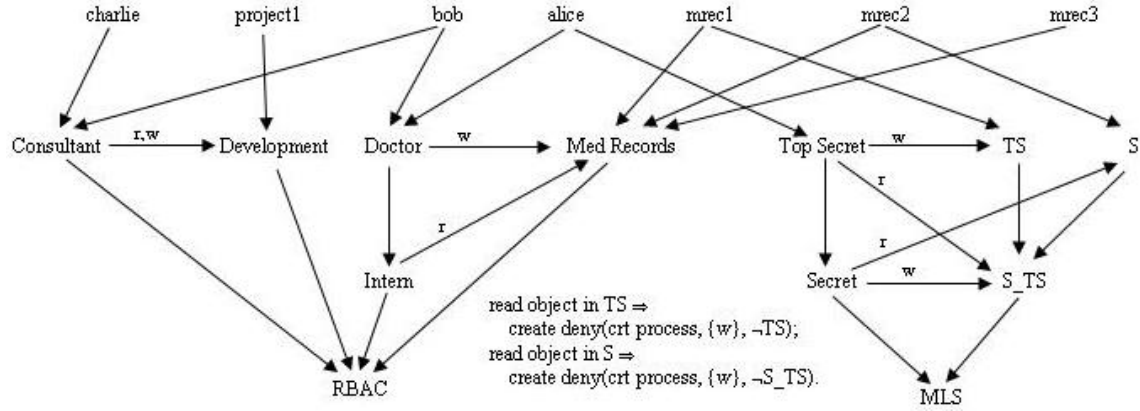
Included in Figure 1 are the PM's basic sets - authorized users ( $U$ ), system operations ( $OP$ ), objects ( $O$ ) and processes ( $P$ ).

Authorized users (hereafter referred to simply as users) are either human beings or "system users". Human users are identified through an authentication mechanism, and system users are identified through the execution of a specific segment of code.

Objects are names (with global meaning) given to system entities that must be protected, and perhaps shared, under one or more access control policies. Among others, Figure 2 shows objects *project1* and *mrec1*. The set of objects may pertain to files, ports, clipboards, email messages, records and fields. The selection of entities included in this set is a matter of choice determined by the protection requirements of the system. The PM maintains the association between the logical object name and the corresponding physical content (e.g., files), or physical object (e.g., system clipboard), as shown in Figure 1.

Operations are actions that can be performed on the contents of objects. Subsets of these operations are environment specific. For example, a PM implementation may include operating system operations such as read (r) and write (w). Also included in the set of PM

operations are a fixed set of administrative operations that create, modify and delete PM data and relations.



**Figure 2. An RBAC-MLS policy combination**

Also included in Figure 1 are user attributes (UA) and object attributes (OA). User and object attributes both characterize users/objects and serve as user/object containers. The set of objects (names) is included in the set of object attributes. User, respectively object, attributes included in Figure 2 are Doctor and Secret, respectively mrec1, Med Records, and TS (for top secret). However, containers can satisfy a variety of uses. Containers may represent folders (e.g., “DHS Proposals”), application-specific work areas (e.g., “Smith Inbox”), or a column in a database table (e.g., “Diagnosis”).

The PM has the ability to configure instances of access control policies, as well as combinations of different access control policies. We denote by *PC* the set of policy classes configured by the PM. Figure 2 includes two policy classes - RBAC and MLS. Also included in Figure 1 is the set of PM processes (*P*). Each PM process is associated with a single user. A user may have multiple PM processes running at any one time. The function *process\_user()* associates a process with its user, i.e., *process\_user(p)* is the user of process *p*.

The following definition summarizes the above discussion.

**Definition 1:** Primitive data sets. The PM primitive data sets are:

- *U* denotes the set of PM users.
- *UA* denotes the set of user attributes.
- *O* denotes the set of PM objects.
- *OA* denotes the set of object attributes, with  $O \subseteq OA$  (each object is an object attribute).
- *OP* is the set of system operations.
- *OPS* is the set of names of the operation sets used in PM.
- *PC* denotes the set of policy classes.
- *P* denotes the set of processes. □

The sets of event pattern/response relations (*EP-R*), user denies (*UDENY*) and process denies (*PDENY*) are described below with respect to the relevant relations.

## 2.2 Assignment relations

The assignment relation is a binary relation between PM entities, denoted by “ $\rightarrow$ ”. There are restrictions on the types of entities for which the assignment relation may hold, as well as other restrictions. This section defines the combinations of entity types for which the assignment may hold.

A user may be assigned to one or more user attributes. For example, in Figure 2 the user with the system id “alice” is assigned to the user attributes “Doctor” and “Top Secret”.

A user attribute may be assigned to another user attribute, as long as the assignment relation remains cycle-free. For example, in Figure 2, the user attribute Doctor is assigned to the user attribute Intern.

An object attribute (and hence an object) may be assigned to another object attribute as long as the second object attribute is not an object and the assignment relation remains cycle-free. For example, in the same Figure 2, the object “mrec1” is assigned to the object attributes “Med Records” and “TS”, and the object attribute TS is assigned to the object attribute S\_TS.

A user attribute may be assigned to a named operation set  $ops$ , where  $ops \subseteq OP$ . In turn, a named operation set may be assigned to an object attribute. In illustrating these assignments, we use abbreviated notations. For example, in Figure 2 we use the notation Doctor  $\xrightarrow{w}$  Med Records instead of Doctor  $\rightarrow \{w\} \rightarrow$  Med Records. As discussed below, these assignment relations indirectly derive the capabilities and permissions, which are fundamental in performing and managing access control.

Not all users (and their processes) are controlled under all policies, nor are all user attributes and object attributes relevant to all policies. To afford appropriate policy class mappings, the user attributes and object attributes may be assigned to relevant policy classes. For example, in Figure 2, the user attribute Intern is assigned to the policy class RBAC and the object attribute “Med Records” is assigned to the same policy class RBAC.

We say that a user or user attribute “belongs to” or “is in” a policy class if there exists a chain of one or more assignments that starts with that user or user attribute and ends with that policy class. Similarly, an object “is protected under” or an object attribute “belongs to” or “is in” a policy class if there exists a chain of one or more assignments that starts with that object or object attribute and ends with the policy class. For example, in Figure 2, the object project1 is protected only under the RBAC policy class (there is an assignment chain project1  $\rightarrow$  Development  $\rightarrow$  RBAC, but there is no assignment starting with project1 and ending with MLS), while the object mrec2 is protected under both RBAC and MLS (mrec2  $\rightarrow$  Med Records  $\rightarrow$  RBAC and mrec2  $\rightarrow$  S  $\rightarrow$  S\_TS  $\rightarrow$  MLS).

No assignment may exist between other types of PM entities than the ones defined above.

The following definition formalizes the above discussion.

**Definition 2:** The assignment relation between PM entities is the union of the following sets of ordered pairs:

- $UUA \subseteq U \times UA$  (assignments user-to-user-attribute).
- $UAUA \subseteq UA \times UA$ , no cycles allowed (assignments user-attribute-to-user-attribute).
- $OA OA \subseteq OA \times (OA \setminus O)$ , no cycles allowed (assignments object-attribute-to-object-attribute).
- $UAOPS \subseteq UA \times 2^{OP}$  (assignments user-attribute-to-operation-set).
- $OPSOA \subseteq 2^{OP} \times OA$  (assignments operation-set-to-object-attribute).
- $UAPC \subseteq UA \times PC$  (assignments user-attribute-to-policy-class).
- $OAPC \subseteq OA \times PC$  (assignment object-attribute-to-policy-class).  $\square$

### 2.3 Conventions

The following conventions are used throughout the remainder of this document. We use the notations  $\rightarrow^*$ , respectively  $\rightarrow^+$  to denote a chain of 0 or more assignments, respectively a chain of 1 or more assignments. For example, consider the assignments  $\text{bob} \rightarrow \text{Doctor} \rightarrow \text{Intern} \rightarrow \text{RBAC}$  of Figure 2. We could write  $\text{bob} \rightarrow^+ \text{RBAC}$  (or  $\text{bob} \rightarrow^* \text{RBAC}$ ). Also, we could write  $\text{bob} \rightarrow^* \text{bob}$  but not  $\text{bob} \rightarrow^+ \text{bob}$ . Also we say that user  $u$  “has a user attribute”  $ua$  if  $u \rightarrow^+ ua$ , and object  $o$  “has an object attribute”  $oa$  if  $o \rightarrow^* oa$ .

### 2.4 Permission Relations

Permissions are assertions regarding the capabilities of users and their processes in performing operations on objects, irrespective of any known exceptions. In their most primitive form, permissions are triples of the form  $(u, op, o)$  indicating that a user  $u \in U$  is able to perform operation  $op \in OP$  on the contents of object  $o \in O$ .

With the assignment relations in place, we can define the PM capabilities and permissions.

**Definition 3:** PM capabilities. The set of capabilities of a user  $u \in U$  in a policy class  $pc \in PC$  is  $\text{caps}_{pc}(u) = \{(op, o) \mid \exists ops \subseteq OP, \exists oa \in OA, \exists ua \in UA : u \rightarrow^+ ua \rightarrow^+ pc, o \rightarrow^* oa \rightarrow^+ pc, ua \rightarrow ops \rightarrow oa, op \in ops\}$  (see Figure 3).  $\square$

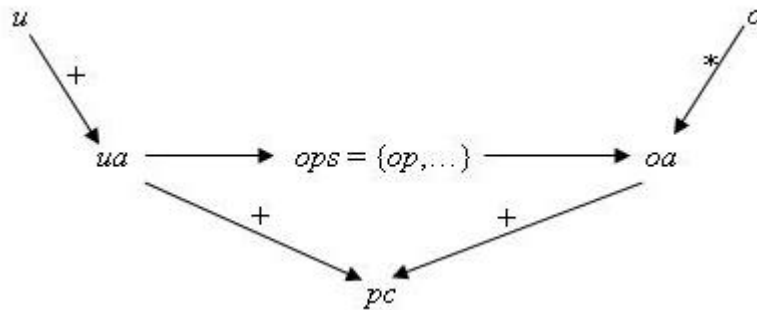


Figure 3.  $(op, o)$  is a capability of  $u$  in  $pc$

Referring to Figure 2, the capabilities of user *alice* in RBAC,  $caps_{RBAC}(alice) = \{(r, mrec1), (w, mrec1), (r, mrec2), (w, mrec2), (r, mrec3), (w, mrec3)\}$  because  $alice \rightarrow Doctor \rightarrow \{w\} \rightarrow Med\ Records$ ,  $alice \xrightarrow{+} Intern \rightarrow \{r\} \rightarrow Med\ Records$ ,  $Doctor \xrightarrow{+} RBAC$ ,  $Intern \xrightarrow{+} RBAC$  and  $mrec1, mrec2, mrec3$  are assigned to *Med Records* and *Med Records*  $\xrightarrow{+} RBAC$ .

**Definition 3:** PM permissions. A triple  $(u, op, o)$  where  $u$  is a user,  $op$  is an operation, and  $o$  is an object, is a PM permission if and only if for each policy class  $pc_k$  under which  $o$  is protected, the pair  $(op, o)$  is a capability of user  $u$  in that policy class  $pc_k$ . In other words, for each policy class  $pc_k$  under which  $o$  is protected, the situation illustrated in Figure 3 holds (see Figure 4): user  $u$  has an attribute  $ua_k$  in  $pc_k$ , object  $o$  has an attribute  $oa_k$  in  $pc_k$ , and there exists an operation set  $ops_k$  containing  $op$ , with  $ops_k$  being assigned to both  $ua_k$  and  $oa_k$ .  $\square$

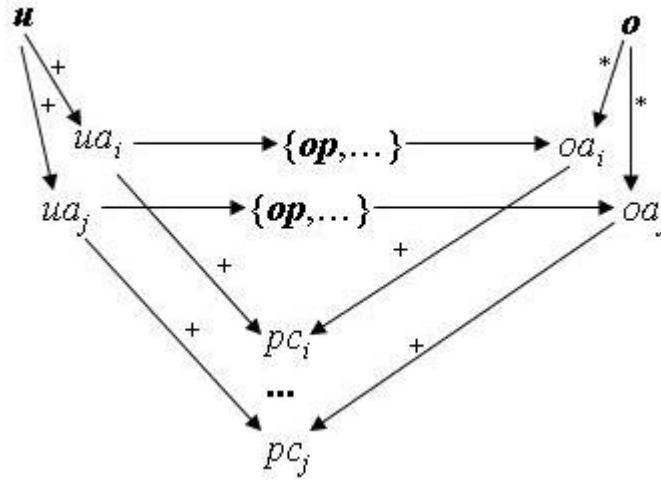


Figure 4.  $(u, op, o)$  is a PM permission

With respect to Figure 2, the triple  $(alice, w, mrec2)$  is a PM permission, because (1)  $mrec2$  is in both RBAC and MLS, (2) both *Doctor* and *Secret* are *alice*'s user attributes, where *Doctor* is in RBAC and  $mrec2$  is in MLS, (3) *S\_TS* and *Med Records* are object attributes of  $mrec2$ , where *Med Records* is in RBAC and *S\_ST* is in MLS, and (4) the operation set  $\{w\}$  is assigned to both *Doctor* and *Med Records* and is assigned to both *Secret* and *S\_TS*. In contrast, the triple  $(bob, w, mrec2)$  is not a PM permission, because  $mrec2$  is in both RBAC and MLS, and *bob* does not have an attribute in MLS.

## 2.5 Administrative operations

An *administrative operation* simply creates, deletes, or modifies an existing policy state data relation. The set of administrative operations include for example create/delete user, create/remove assignment, etc. The state of the overall PM policy changes as a consequence of the execution of an administrative operation. The administrative operations are executed on behalf of a user via administrative commands (users never execute operations directly - see below), or automatically by the PM in response to a recognized event. Event-response relations are described in the following section.

An administrative operation could be specified as a parameterized procedure, whose body describes how a data set or relation (denoted by  $R$ ) changes to  $R'$ :

$$\begin{array}{l} opname(x_1, \dots, x_k) \{ \\ \quad R' = f(R, x_1, \dots, x_k) \\ \} \end{array}$$

For example, consider the following administrative operation CreateUser:

$$\begin{array}{l} CreateUser(u) \{ \\ \quad U' = U \cup \{u\} \\ \} \end{array}$$

The CreateUser administrative operation specifies that the creation of a new user with the id  $u$  consists of augmenting the user set  $U$  with the new user id. Included in this specification is the fact that if a user with the same id already exists, the operation has no effect.

An *administrative command* is a parameterized sequence of administrative operations prefixed by a condition and has the format:

$$\begin{array}{l} commandName(x_1, \dots, x_k) \\ \quad \text{if } (condition) \text{ then} \\ \quad \quad paop_1 \\ \quad \quad \dots \\ \quad \quad paop_n \\ \quad \text{end} \end{array}$$

where  $x_1, \dots, x_k$  ( $k \geq 0$ ) are (formal) parameters and  $paop_1, \dots, paop_n$  ( $n \geq 0$ ) are primitive administrative operations which may use  $x_1, \dots, x_k$  as their parameters. The condition tests, in general, whether the user who requested the execution of the command is authorized to execute the command (i.e., the composing primitive operations), as well as the validity of the actual parameters. If the condition evaluates to false, then the command fails. For example, the command that grants a user attribute a set of operations on an object container could be defined as follows:

$$\begin{array}{l} grant(cert\_process, ua, oa, ops, op_1, \dots, op_m) \\ \quad \text{if } (ua \in UA \wedge oa \in OA \wedge \\ \quad \quad \forall i \in 1..m \ op_i \in OP \wedge \\ \quad \quad ops \notin OPS \wedge \\ \quad \quad is\_auth(cert\_process.user, create\_opset) \wedge \\ \quad \quad is\_auth(cert\_process.user, oattr\_assign\_opset\_to, oa) \wedge \\ \quad \quad is\_auth(cert\_process.user, uattr\_assign\_to\_opset, ua)) \text{ then} \\ \quad \quad create\_opset(ops, op_1, \dots, op_m) \\ \quad \quad assign\_opset\_to\_attr(ops, oa) \\ \quad \quad assign\_attr\_to\_opset(ua, ops) \\ \quad \text{end} \end{array}$$

For convenience, a command may exist inside another command.

## 2.6 Prohibitions

Permission relations alone are not sufficient in specifying and enforcing the current access state for many types of policies. Other policies pertain to prohibitions or exceptions to permissions. Deny relations specify such prohibitions. PM deny relations take on two forms – user-based deny and process-based deny. User-based deny relations associate users with capabilities ( $op, o$ ) that the user and the user's subjects are prohibited from executing. For example, although a user with the attribute IRS Auditor may be allowed to review IRS tax records, a user-based deny relation could prevent that user from reviewing his/her own tax record. Process-based deny relations associate processes with capabilities ( $op, o$ ) that the processes are prohibited from executing. Process-based deny relations are usually created through the use of obligations (see obligations below). User-based deny relations can be created either through administrative commands or through obligations.

**Definition 4.** A user-based deny relation is a triple  $\langle u, ops, os \rangle$ , where  $u \in U$ ,  $ops \subseteq OP$ , and  $os \subseteq O$ . Its meaning is that a process executing on behalf of user  $u$  cannot perform any of the operations in  $ops$  on any of the objects in  $os$ . We denote by  $UDENY$  (see Figure 1) the set of user-based deny relations.  $\square$

**Definition 5.** A process-based deny relation is a triple  $\langle p, ops, os \rangle$ , where  $p \in P$ ,  $ops \subseteq OP$ , and  $os \subseteq O$ . Its meaning is that the process  $p$  may not perform any of the operations in  $ops$  on any of the objects in  $os$ . We denote by  $PDENY$  (see Figure 1) the set of process-based deny relations.  $\square$

## 2.7 Obligations

An obligation or *event pattern/response* relation defines a set of conditions and methods under which policy state data is dynamically obligated to change. An event pattern/response relation is a pair ( $ep, r$ ) (usually denoted  $ep \Rightarrow r$ ), where  $ep$  is an *event pattern* and  $r$  is a sequence of primitive administrative operations, called a *response*. The event pattern specifies conditions related to a process' successful execution of an operation on an object, using parameters like the user of the process, the operation executed, and the container(s) in which the object is included. We denote the set of event pattern/response relations by EP-R (see Figure 1).

A successful completion of an operation on an object may trigger an event. The context of the event comprises the process id and its user identity, the operation, the object on which the operation was performed, the object's containers, etc. The PM starts processing the event by determining the event patterns matched by the event. The match is performed by checking whether the event context satisfies the conditions specified in the event pattern. For all successful matches, the PM executes the response associated with the matched pattern. Note that the possible formal parameters of the administrative operations comprised in the response are replaced by the appropriate values extracted

from the event context. Responses are obligations performed by the PM, and as such, their execution is not predicated on permissions.

### 3 PM FUNCTIONS

Included within the PM are a fixed set of functions. These functions include user authentication, session management, presentation of accessible objects, reference mediation, and event-response processing.

#### 3.1 Authentication

A user interaction with the PM begins with the user's authentication. Although authentication is included among the PM's functions, the PM's specification does not dictate the method (e.g., password, tokens) by which authentication is performed. Upon authentication, a session is created where all processes included in the session are associated with the authenticated user.

A process may issue an access request on behalf of its user, or independent of its user. We denote by  $\langle ops, o \rangle_p$  a process access request, where  $p \in P$ ,  $ops \subseteq OP$  and  $o \in O$ .

#### 3.2 Personal object system (POS)

Following user authentication, the user may be presented with a Personal Object System (POS). The POS is a graphical presentation of the set of objects that are currently accessible to the user. The graphical presentation organizes accessible objects into the set of containers (object attributes) to which the objects belong and which are also accessible by the user. Remembering that objects are also object containers, an object container is accessible to a user if that user is authorized to perform the *same* operation  $op$  on the objects contained in that container within *each* policy class that container belongs to. Formally,

**Definition 6:** Accessible object container. An object container (or attribute)  $oa \in OA$  is accessible to a user  $u \in U$  if  $\exists op \in OP$ , such that  $\forall pc \in PC$  such that  $oa \rightarrow^+ pc$ ,  $\exists ua \in UA$ ,  $\exists ops \subseteq OP$ ,  $\exists oa' \in OA$ , such that  $op \in ops$ ,  $u \rightarrow^+ ua \rightarrow ops \rightarrow oa'$ ,  $ua \rightarrow^+ pc$ , and  $oa \rightarrow^* oa' \rightarrow^+ pc$  (see Figure 5).  $\square$

Now we can formally define the POS of a user  $u$  as a graph whose nodes and edges are defined through their property.

**Definition 7:** Personal object system. The personal object system of a user  $u$ , denoted by  $POS_u$ , is a directed graph  $(V, E)$  where the node set  $V$  is defined as follows:

1. An object attribute  $oa$  is in  $V$  if and only if  $oa$  is in a policy class and  $oa$  is accessible to user  $u$ .
2. A policy class  $pc$  is in  $V$  if and only if  $pc$  contains an object attribute accessible to  $u$ .
3. No other node is in  $V$ .

and the arc set  $E$  is defined as follows:

4. If policy class  $pc$  and object attribute  $oa$  are nodes in  $V$ , there is an arc from  $oa$  to  $pc$  if and only if  $oa$  is in  $pc$  (in the original graph) and there is no other object attribute in  $V$  on a path from  $oa$  to  $pc$  (in the original graph).
5. If  $oa_1$  and  $oa_2$  are nodes in  $V$ , there is an arc from  $oa_1$  to  $oa_2$  if and only if there is a path from  $oa_1$  to  $oa_2$  (in the original graph) and there is no other object attribute in  $V$  on a path from  $oa_1$  to  $oa_2$  (in the original graph).
6. No other arc is in  $E$ .  $\square$

Figure 6 is a representation of alice's POS with respect to Figure 2.

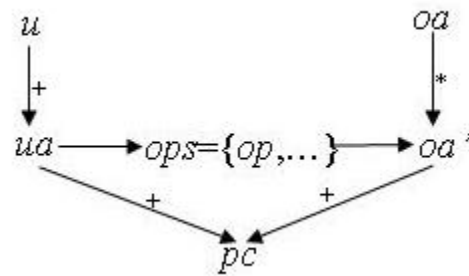


Figure 5. Container  $oa$  is accessible to user  $u$ : for each  $pc$  such that...

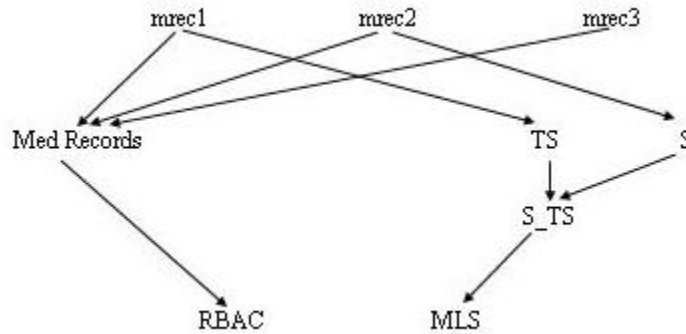


Figure 6. Alice's Personal Object System

### 3.3 Reference mediation

Either through the use of the POS or some other means of referencing objects, a user may issue a request to perform a set of operations on a set of objects, through a process. A process may issue an access request without the intervention of a user. The PM either grants or denies a process access request. A process access request to perform an operation  $op$  on an object  $o$ , with  $p$  being the process identifier, is granted if and only if there exists a PM permission  $(u, op, o)$  where  $u = process\_user(p)$ , and  $(op, o)$  is not denied for either  $p$  or  $u$ . We refer to this function of granting or denying a process access request as *reference mediation*.

**Definition 8.** Given process  $p \in P$ , user  $u = process\_user(p)$ , operation  $op \in OP$ , and object  $o \in O$ ,  $reference\_mediation(<op, o>_p) = grant \stackrel{def}{\Leftrightarrow}$

1.  $(u, op, o)$  is a PM permission  $\wedge$
2.  $\forall <u, ops, os> \in Udeny, \neg(op \in ops \wedge o \in os) \wedge$
3.  $\forall <p, ops, os> \in Pdeny, \neg(op \in ops \wedge o \in os). \square$

With respect to the definition of PM capabilities and permissions, we can note that the reference mediation function grants a process  $p$  the permission to execute a request  $<op, o>_p$  if and only if, in *each* policy class *that contains the object*  $o$ , the pair  $(op, o)$  is a capability of an attribute of user  $u = process\_user(p)$ , and in addition, this capability is not prohibited by a deny relation.

### 3.4 Transferring data between processes

Underlying resource management systems, on which access control depends, provide facilities for inter-process communication, and as such offer opportunities to “leak” data in a manner that may undermine the policy. For example, operating systems provide mechanisms for facilitating communications and data sharing between applications. These mechanisms include but are not limited to clipboards, pipes, sockets, remote procedure calls, and messages. They all conform to a common abstraction: one process produces/creates data and inserts it into the mechanism’s physical medium; the other process consumes/reads the data from the physical medium. A synchronization mechanism must also exist.

By treating the communication medium as a PM object, the PM offers strategies to support data transfer that are in compliance with the policy. For example, the producer process could create a PM object that represents the physical medium/support of the data transfer mechanism. This new object will be assigned attributes in accordance to a predefined and policy-specific set of conditions. These conditions can be specified via the event-response relations (e.g., if a process reads secret data, any subsequently created object will be assigned to the secret attribute). The consumer process must be able to read the PM object that represents the physical medium under the rules of the reference mediation.

A practical example pertains to the system clipboard that is used in performing copy/cut and paste operations. When process  $p1$  issues the copy/cut request, the PM creates an object, say  $co$ , which represents the clipboard, with attributes that are specified in the event-response relations. The data is transferred to the system clipboard as usual. When the same or different process  $p2$  issues the paste request, the PM naturally treats this request as a request to read from object  $co$ . As for any other process request, the PM invokes the reference mediation function to check whether  $p2$  is authorized to read the object  $co$ . If the request is granted by the reference mediation function,  $p2$  continues with the paste operation as usual.

## 4 ILLUSTRATIVE EXAMPLES

In this section, we illustrate the PM's ability to support the security objectives of RBAC and MLS security models. It is important to note that when we say that the PM is able to support a particular policy objective we are not necessarily suggesting that we emulate the specific rules or relations of any model, but rather that the PM is able to implement the same policy objectives of those models through PM configuration.

### 4.1 Combining RBAC and MLS

Consider the combination of the two access control policies depicted in Figure 2, a role-based access control (RBAC) policy, and a multi-level security (MLS) policy.

#### 4.1.1 RBAC

The main administrative objective of RBAC, as suggested by various RBAC models [9] and the RBAC standard [6], is to streamline authorization management by defining roles as relations between users and capabilities. These relations are achieved by assigning users to roles on one side and assigning capabilities to roles on the other side. By assigning a user to a role, that user instantaneously acquires the capabilities that are assigned to the role.

Another RBAC feature is the ability to define a role hierarchy, i.e., an inheritance relation between roles, whereby senior roles acquire the capabilities of their juniors. By assigning a user to a role, the user is also (indirectly) associated with the capabilities of that role's junior roles.

Finally, the RBAC standard includes two types of relations for the enforcement of separation of duties – static separation of duty (SSoD) and dynamic separation of duty (DSoD).

The Policy Machine meets these administrative and policy objectives of RBAC. Indeed, a PM user attribute includes the semantics of a RBAC role; by assigning a user to that user attribute, the user acquires the capabilities associated with the user attribute. Moreover, PM is superior to RBAC in administrative efficiency, due to additional abstractions. In PM, capabilities are indirectly associated with user attributes through the double assignment user attribute – operation set – object attribute. By assigning a user to a user attribute the user is capable of performing the operations in the operation set on the objects in the container represented by the object attribute. Furthermore, PM allows for the efficient association of objects with access control entries of the form (*user*, *operation*), while RBAC offers no semantics in this regard. With regard to role hierarchies, the PM offers semantics similar to RBAC through the user attributes assignments to other user attributes. Finally, the PM allows for the inheritance of access control entries between object attributes.

Consider the PM permissions configuration illustrated by Figure 2. The user attributes Doctor, Intern, and Consultant represent RBAC roles. Included in the configuration are object attributes Med Records and Development, and objects like *mrec1* and *project1*. Under the RBAC policy, user *alice*'s permissions are directly derived from *alice*'s assignment to the user attribute Doctor ( i.e.,  $(alice, w, mrec1)$ ,  $(alice, w, mrec2)$ , and  $(alice, w, mrec3)$ ). *Alice* also inherits the permissions  $(alice, r, mrec1)$ ,  $(alice, r, mrec2)$ ,  $(alice, r, mrec3)$  from the assignment *Doctor*  $\rightarrow$  *Intern*, which offers the same semantics as that of the role hierarchy.

Conflict of interest in a role-based system may arise as a result of a user gaining authorization for capabilities associated with conflicting roles. One means of preventing this form of conflict of interest is through static separation of duty (SSD), that is, to enforce constraints on the assignment of users to roles. SSD relations place constraints on the assignments of users to roles. Membership in one role may prevent the user from being a member of one or more other roles. Dynamic separation of duty (DSD) relations, like SSD relations, limits the capabilities that are available to a user. However DSD relations differ from SSD relations by the context in which these limitations are imposed. DSD requirements limit the availability of the capabilities by placing constraints on the roles that can be activated within or across a user's sessions.

The PM is able to meet the same security objectives as SSD and DSD, but through different means. Assume that a conflict of interest would arise if a user were able to execute capability  $(op1, o1)$  and capability  $(op2, o2)$ . Under RBAC, these capabilities would be assigned to different roles (say *r1* and *r2*) and an SSD relation would be imposed between those roles and thus prevent any user from being simultaneously assigned to both roles. The following PM obligation relation can be used to achieve this same objective:

process performs  $(op1, o1) \Rightarrow \text{deny}(\text{process user}, \{op2\}, o2)$   
process performs  $(op2, o2) \Rightarrow \text{deny}(\text{process user}, \{op1\}, o1)$

Through these relations any process that successfully executes  $(op1, o1)$  would effectively deny the process user the ability to successfully execute  $(op2, o2)$  in the future and vice-versa. Furthermore, in an RBAC SSD environment, while a user *u1* that is assigned to *r1* would be prevented from executing  $(op2, o2)$  through denial of membership to *r2*, nothing prevents  $(op2, o2)$  from being assigned to some *r3* and *u1* being assigned to *r3*.

Regarding DSD, assume once again the capability  $(op1, o1)$  and capability  $(op2, o2)$  that are respectfully assigned *r1* and *r2*. Also, assume *r1* and *r2* are in a DSD relation in RBAC. Under these circumstances, no user may have the ability to execute both capabilities within the same session. However, a user can assume both roles in different sessions either concurrently or sequentially. Given that DSD does not limit a user's ability to execute both capabilities within different session, we see the security objective as being that of enforcement of least privilege at the process level. Under least privilege a process should be prevented from executing (either maliciously or by error) both

capabilities. The following PM obligation relation can be used to achieve this same objective:

process performs ( $op1, o1$ )  $\Rightarrow$  deny (current process,  $\{op2\}, o2$ )  
process performs ( $op2, o2$ )  $\Rightarrow$  deny (current process,  $\{op1\}, o1$ ).

#### 4.1.2 MLS

The objective of the MAC [1] security policy is to prevent the unauthorized reading of classified information. Traditionally, this policy objective has been specified and often implemented in terms of the simple security and \*-property of the Bell & LaPadula [2] security model, and is often referred to as the Multiple Level Security (MLS) policy. Under the MLS policy, security levels, organized under a dominance relation ( $\geq$ ), are assigned to subjects (users and their processes) and objects. The simple security property specifies that a subject is permitted read access to an object only if the subject's security level dominates the object's security level, and the \*-Property specifies that a subject is permitted write access to an object only if the object's security level dominates the subject's security level. Indirectly, the \*-Property prevents the transfer of data from an object of a higher level to an object of a lower classification. The security objective of these two rules is to prevent the direct and indirect reading of information at a level higher than the user's level.

Figure 2 is an illustration of a PM configuration that meets these security objectives in terms of permission relations, and obligation relations. Figure 2 assumes top-secret  $\geq$  secret. The permission relations of Figure 2 specify that users cleared to the levels of top-secret and secret are respectively assigned to the Top Secret and Secret user attributes, and objects that are classified at the top-secret and secret levels are respectively assigned to the TS and S object attributes. The S\_TS object attribute is a container for all objects classified at top-secret or secret levels. With respect to these permission relations alone, users (and their processes) that are assigned to Top Secret are only able to perform read operations on objects classified at the levels top secret and secret and users (and their processes) that are cleared secret are only able to perform read operations on objects classified at the level secret, thus showing support for the security objectives of the simple security property.

However, under these permission relations, a user like *alice*, for example, could read top secret data and subsequently write that data to a secret object. To prevent such leakage of classified information, the PM configuration of figure 2 comprises two event-response relations:

- (1) read *TS* object  $\Rightarrow$  create deny(current process,  $\{w\}, \neg TS)$ ;
- (2) read *S* object  $\Rightarrow$  create deny(current process,  $\{w\}, \neg S\_TS)$ .

The first relation specifies that whenever a process successfully reads a top-secret object, it will be denied the ability to write to objects that are not in the TS container (the “ $\neg$ ” symbol stands for “the complement of”). The second relation specifies that whenever a

process successfully reads a secret object, it will be denied the ability to write to objects that are not in the S\_TS container (i.e., neither S nor TS).

Under our strategy and configuration, a process with its user cleared to a particular level (say top secret), can read objects at levels at or below the clearance level of the user (i.e., top secret, or secret). However, once a process has read data at a particular level (say top secret), that process can no longer write to objects below that particular level (i.e., secret). These observations demonstrate adherence to the policy objective of both the simple security and \*-property of the Bell & LaPadula security model.

#### **4.1.3 RBAC and MLS combination**

Under the combination of MLS and RBAC policy instances, only processes with associated users that are Interns and Doctors and are cleared to the top secret level (e.g., alice) may perform both read and write operations on objects that are Med Records and are classified TS at the same time (e.g., *mrec1*).

Assume that user alice opens a PM session by authenticating herself to the PM. As mentioned above, alice is presented with her POS, as depicted in Figure 5. Further assume that alice issues a request to open the object *mrec1* for reading and writing in a process executing on her behalf. PM determines that *mrec1* is protected under both RBAC and MLS policies, and that alice is authorized to access *mrec1* in both policies, through her attributes Intern and Doctor of RBAC and Top Secret of MLS. The process request to read *mrec1* is granted by the reference mediation function. After modifying the memory image of *mrec1*, alice may issue a request through her process to save (write) *mrec1*'s contents. The request issued through that process is granted based on the same considerations as before.

The following discussion shows how a user is prevented from leaking information from a higher security level (e.g., TS) to a lower level (e.g., S) through cut/copy and paste operation with respect to the above policy configuration.

Assume the following event-response relations in addition to those defined above:

- (3) read TS object  $\Rightarrow$   
create event-response(current process create object  $\Rightarrow$   
assign new object to TS);
- (4) read S object  $\Rightarrow$   
create event-response(current process create object  $\Rightarrow$   
assign new object to S);
- (5) copy object  $\Rightarrow$  assign clipboard(current host) to attributes(current object).

The effect of relation (3) is that if a process successfully reads a top-secret object, whenever that process subsequently creates an object, the new object will be assigned to the TS attribute. The effect of relation (4) is that if a process successfully reads a secret object, whenever that process subsequently creates an object, the new object will be

assigned to the S attribute. The effect of relation (5) is that if a clipboard operation “copy” is performed on a source object, the object that represents the clipboard is assigned to the attributes of the source object.

**a) In the same process.** Assume that user alice issues a request to read *mrec1* (TS) in a process *p*. The reference mediation function grants the request. At the successful completion of the read operation, a deny relation  $(p, \{w\}, \neg TS)$  is added to the policy configuration as specified by the event-response relation (1). Also, an event-response

(3.1) *p* creates object  $\Rightarrow$  assign new object to *TS*

is generated according to (3). Next, alice issues a request to read *mrec2* (S) in the same process *p*. The reference mediation function again grants the request. At the successful completion of the read operation, a deny relation  $(p, \{w\}, \neg S\_TS)$  is added to the policy configuration as specified by the event-response relation (2). Also, an event-response

(4.1) *p* creates object  $\Rightarrow$  assign new object to *S*

is generated according to (4).

Now alice tries to copy some information from object *mrec1* (TS) to object *mrec2* (S) and save the latter. To copy the information from object *mrec1* to the clipboard, the process *p* first creates a PM object that represents the clipboard. According to (3.1) and (4.1) the new object is assigned to both TS and S. Second, the process *p* actually copies the information from *mrec1* to the clipboard and a “copy object” event is generated. According to relation (5), the clipboard object is assigned to all attributes of *mrec1*. The fact that the clipboard object is already assigned to TS does not matter. Hence, the PM clipboard object becomes assigned to TS, S, and Med Records.

Next, alice pastes the clipboard content to the *mrec2* object. The paste action starts with a read operation from the clipboard object, which is classified TS and S. According to the event-response relations (1) and (2), the PM generates the deny relations  $(p, \{w\}, \neg TS)$  and  $(p, \{w\}, \neg S\_TS)$ . The clipboard content is pasted into the *mrec2* object. Finally, alice tries to save (write) the *mrec2* object. Because *mrec2* is not contained in TS, one of the deny relations  $(p, \{w\}, \neg TS)$  prevents the current session from saving *mrec2*.

When alice tries the reverse operation, namely to copy some information from object *mrec2* (S) to object *mrec1* (TS) and save the latter object, it is easy to show (applying the same kind of reasoning) that she succeeds.

**b) In different processes.** User alice issues a request to read *mrec1* (TS) in a process *p1*. The reference mediation function grants the request. At the successful completion of the read operation, a deny relation  $(p1, \{w\}, \neg TS)$  is added to the policy configuration as specified by the event-response relation (1). Also, an event-response

(3.2) *p1* creates object  $\Rightarrow$  assign new object to TS

is generated according to (3). Next, alice issues a request to read *mrec2* (S) in a new process *p2*. The reference mediation grants the request. At the successful completion of the read operation, a deny relation ( $p2, \{w\}, \neg S\_TS$ ) is added to the policy configuration as specified by the event-response relation (2). Also, an event-response

(4.2) *p2* creates object  $\Rightarrow$  assign new object to S

is generated according to (4).

Let's assume that alice tries to copy some information from object *mrec1* (top-secret) to object *mrec2* (secret) and save the latter. To copy the information from object *mrec1* to the clipboard, the process *p1* first creates a PM object that represents the clipboard. According to (3.2) and (4.2) the new object is assigned to TS. Second, the process *p1* actually copies the information from *mrec1* to the clipboard and a "copy object" event is generated. According to relation (5), the clipboard object is assigned to all attributes of *mrec1*. The fact that the clipboard object is already assigned to TS does not matter. Hence, the PM clipboard object becomes assigned to TS and Med Records.

Next, alice pastes the clipboard content to the *mrec2* object in process *p2*. The paste action starts with a read operation from the clipboard object, which is classified TS. According to the event-response relations (1), the PM generates the deny relations ( $p2, \{w\}, \neg TS$ ). The clipboard content is pasted into the *mrec2* object. Finally, alice tries to save (write) the *mrec2* object. Because *mrec2* is not contained in TS, the deny relation ( $p2, \{w\}, \neg TS$ ) prevents the current session from saving *mrec2*.

## 4.2 Preventing data leakage in RBAC

RBAC is not designed to prevent unauthorized leaking of data. For example, with respect to Figure 2, the RBAC policy specifies that doctors and interns can read medical information, and this suggests to many that only doctors and interns can read medical information. Under this configuration, nothing prevents bob from copying the contents of *mrec3* and pasting it into the object *project1*, which can be read by charlie who is not a Doctor or Intern. It should be noted that a malicious process acting on bob's behalf could also read medical information and write it to *project1* without bob's knowledge.

To prevent this unlawful leakage, we can apply the approach that was used to prevent leakage under MLS to the context of RBAC. Consider the following event-response relation:

(6) read "Med Records" object  $\Rightarrow$   
create deny(current process,  $\{w\}, \neg$ "Med Records").

Relation (6) will prevent bob using a single process from reading contents of any medical record (e.g., *mrec3*) and subsequently writing it to any object outside the Med Records container (e.g., *project1*).

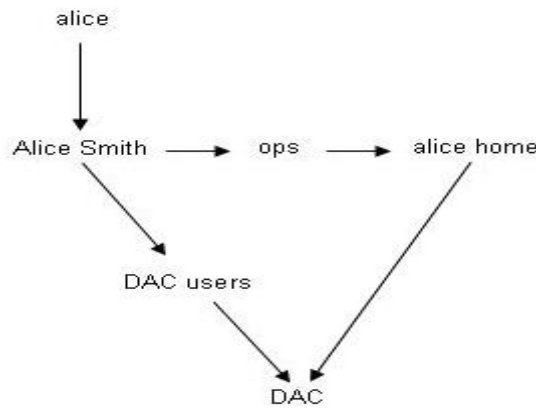
In a scenario where bob copies data from *mrec3* and pastes it to *project1* in different processes, relation (5) assigns the clipboard object to the Med Records container. The

second process for the paste operation reads the clipboard object, and according to the relation (6) the PM generates a deny relation that prevents bob from writing (saving) *project1*.

### 4.3 Discretionary access control

Under Discretionary Access Control (DAC) [1], the user who creates an object is called the object “owner” and controls users’ capabilities on that object based on the users’ or user groups’ identities. The capabilities that the owner controls include operations on the object’s content (e.g., read/write/execute), as well as operations that change the object’s access control policy (e.g., transfer ownership of the object or grant/revoke users’ access to the object).

PM provides support for the configuration of DAC policies. For example, the user’s identity can be represented through a user attribute that specifies the name or identity of the user and which has that user as its only member (i.e., the user in question is the only user assigned to this user attribute). We call this attribute the “name attribute”, or the “identity attribute”. Similarly, a group identity could be specified as a user attribute that contains only the users that are members of that group. In Figure 7, which partially illustrates a PM DAC configuration, the user attribute “Alice Smith” is user alice’s name attribute, while the “DAC users” user attribute represents the group of all users included in the DAC policy class.



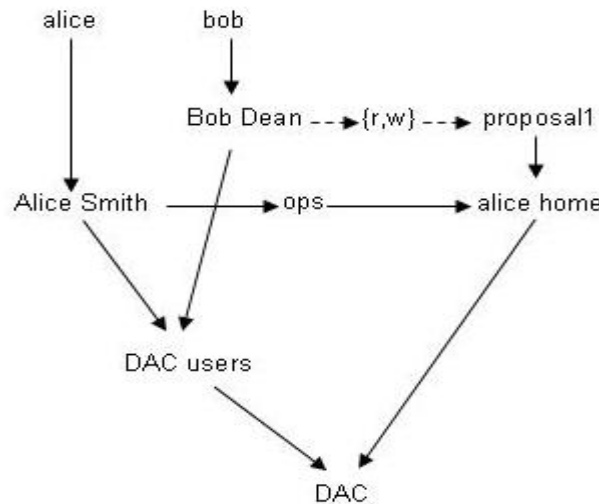
**Figure 7. A partial DAC configuration**

User’s ownership and capabilities over an “owned” object can be specified under this PM configuration by placing the object in a container specially created for that user. We refer to this container as the user’s home. In figure 7, for example, the object attribute “alice home” denotes the home container of user alice. The creation of a user’s home must be accompanied by setting up three categories of capabilities for the user: (a) capabilities to access the content of the objects contained in the home container; (b) capabilities to perform administrative operations on the contents of the home container (e.g., object attribute to object attribute assignments, creation of new object attributes); and (c) capabilities to transfer ownership or grant/revoke other users’ access to the objects inside the home container. The user, his/her home container and the capabilities (a), (b), and (c) could be conveniently created through a single administrative command –

*create\_dac\_user* (*user id*, *user name*). Typically, under DAC a user initially obtains ownership and control over an object as a consequence of object creation. This can be achieved by the PM by defining an event-response relation where the event is the object creation and the response is the assignment of the new object to the user's home container.

Using the policy configuration described above, transferring the ownership of an object to another user may be achieved by assigning the object to the other user's home container and optionally deleting its assignment to the original owner's home. Note that the transfer requires the permission to assign objects from the original owner home to another user's home container.

Granting another user or group of users access to an object *o* may be achieved by the owner by creation of the assignment  $g \rightarrow \{r, w\} \rightarrow o$  where *g* is a user attribute that represents the other user or group of users in the DAC users. Figure 8 shows how alice could grant user bob read/write access to one of her objects by using such assignments to bob's name attribute "Bob Dean". Other configuration strategies exist as well.



**Figure 8. Alice grants bob read/write access to proposal1**

#### 4.4 Library of policy configurations

An additional feature of the PM is the capability to establish a library of policy configurations. The principle is that an administrator does not need to configure policy from scratch. Once a policy (say DAC) has been defined and tested by security experts, the policy can be made available for importation and as such the policy becomes instantiated. Policy configuration can also be parameterized, providing opportunities for customization. For example, with respect to an MLS policy, the elements of the dominance relation would be parameterized, and the specific levels could be defined just prior to importation, or with respect to a DAC policy, delegation details could be defined.

## 5 General Architecture

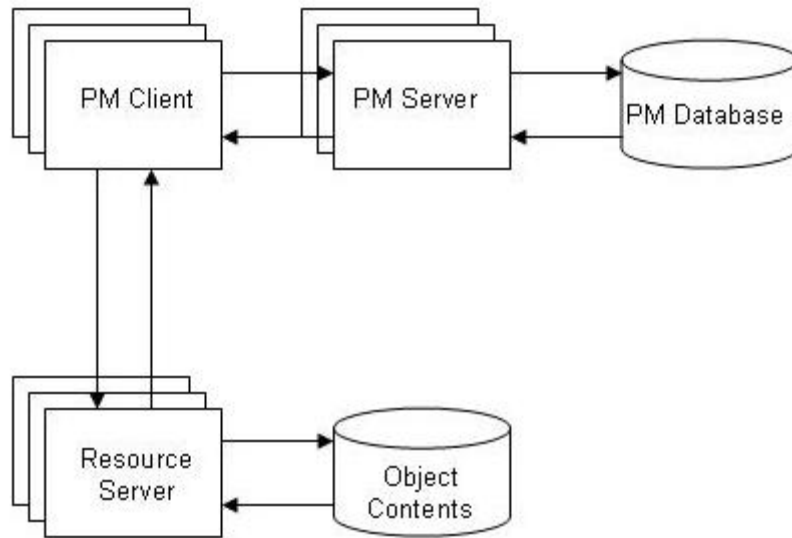
In its simplest form, the general architecture of the PM comprises one or more *PM clients*, one or more *PM servers*, a *PM database*, and one or more resource servers (see Figure 9).

The PM client or the *user environment* is the context in which the user's PM processes run. It can be an operating system, an application (e.g., a database management system), a service in a service oriented architecture, or a virtualized environment. Typically, a PM user logs on the PM by using a GUI provided by the PM client. A successful login opens a user session on the PM client. Within the session, the user may create and run various PM processes that request access to PM-protected resources. The PM client traps each access request, and then asks a PM server to decide whether to grant or reject the access request. Because the physical location of each object is known to the PM server (and transparent to the client), in the case of a granted request, the server response is accompanied by the physical location of the object. The PM client next enforces the server's decision, granting or rejecting the process' access to the object. For a granted request the PM client requires the cooperation of a *resource server* in performing the granted operation on the physical content of an object. The resource server can reside on a machine that includes the user environment or on a server that is dedicated the storage of PM resources.

The PM client exposes a standard set of APIs that can be used to develop PM-aware applications.

The PM server or *decision maker* is a software module that receives an access request from a PM client, computes a decision to grant or reject the access request, and returns the result. In this aspect, the PM server implements PM's reference mediation function. The decision is based on the identity of the user/process that issued the request, the requested operation, the requested resource/object, and the permission and prohibition relations as stored in the PM database. The PM server also executes the responses to events specified in the obligation relations stored in the PM database, when the events are triggered by a client's successful execution of an operation on a PM object. Finally, the PM server can be used to administer the PM database. The PM server exposes a standard set of commands that can be used by clients to solicit its services.

The PM database contains a standard set of data and relations that represent the current policy configuration.



**Figure 9. General PM architecture**

Another component not illustrated in Figure 9 is PM's *resource repositories*, where PM stores the physical contents of its objects. The repository of each object is obviously known to the PM, but transparent to the user. The PM client is responsible for transferring the contents of an object to the PM user environment where the requesting process resides.

## 6 PM Applications

There are two types of PM applications. The first type comprises those applications that provide services that are independent to access control. These applications include, for example, text editors, spreadsheets, and drawing packages. The second type comprises those applications that provide services through the implementation of some access control policy. For example, e-mail applications provide for the reading of messages and attachments through the discretionary distribution of objects, and workflow management applications provide for the reading and writing of specific documents by a prescribed sequence of users.

### A simple workflow application

In this section, we discuss a PM configuration that supports a simple workflow application. Let us consider a workflow example comprising the following activities and users or roles that must perform those activities sequentially.

**Activity 0.** A user in the “Secretary” role fills out a purchase order form and attaches a “routing slip” that specifies  $n \geq 1$  users and/or roles and the order in which they must approve and sign the purchase order.

**Activity  $k = 1$  to  $n$ :** The user or a user in the role specified in the routing slip at position  $k$  approves and signs the purchase order.

**Activity n+1:** A user in the “Acquisition” role examines the purchase order before ordering the items.

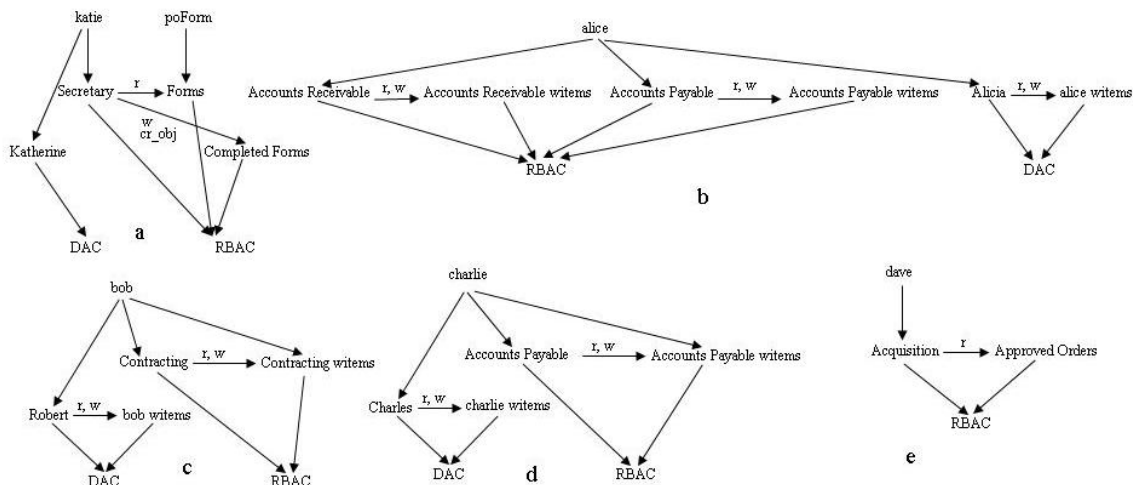
We assume that the workflow policy also imposes the restriction “No user is allowed to sign a purchase order twice”.

In the following, we describe the PM configuration used to specify and enforce the policy described above.

First, the purchase order will be modeled by a PM object. Activity 0 performed by a user in the Secretary role consists of reading an empty form object, filling out the form and creating a purchase order object with the data from the form. Each signing activity consists of reading the purchase order object from the specified user’s or role’s work items, applying maybe a graphic and/or electronic signature to its content, and writing back the purchase order object. Finally, activity n+1 consists of simply reading the purchase order object. The purchase order object will not be accessible to a user unless all previous activities as specified in the sequence have been successfully completed. Note that the policy enforcement will be performed by the OS kernel, not by the application.

The PM configuration will include two policy classes, DAC and RBAC. The DAC policy will comprise the user ids, the user name attributes, and a work items object container for each user. Each user has read/write access to its work items.

The RBAC policy will comprise the user ids, the Secretary role, a few “signing” roles, and the Acquisition role. The Secretary role has read access to a blank purchase order form included in the Forms container, write access to a container of Completed Forms, and the privilege of composing and registering event-response relations with the PM. Each signing role has read/write access to its work items. The Acquisition role has read access to the container of Approved Orders. Figures 10 a)-e) illustrates a configuration with three signing roles (Accounts Receivable, Contracting, and Accounts Payable) and three signing users (alice, bob, and charlie), a Secretary user, katie, and an Acquisition user, dave.



**Figure 10. Policy configuration for a simple workflow application**

The activity sequencing will be ensured by the PM changing the purchase order location after each successful completion of an activity. Behind the automatic moves performed by the PM is an event-response script composed and registered with the PM by the workflow application running on behalf of the user acting in the Secretary role, *just before the creation* of a new purchase order.

Processing of a purchase order starts with the user katie in the Secretary role filling the empty form, attaching a routing slip, and saving the form in the “Completed Forms” container as object po121 for example. The same user also generates  $n+1$  event-response relations that specify what should happen after the successful completion of each of the activities  $0, 1, \dots, n$ .

For example, assuming that the routing slip as composed by katie contains, in order, user alice, role Contracting, and role Accounts Payable. The event-response relation corresponding to the successful completion of activity 0 might look as follows, assuming that the first on the routing slip is user alice:

$R_0$ : write object po121 in “Completed Forms”  $\Rightarrow$   
    assign crt\_object to “alice work items”;  
    delete assignment of crt\_object to “Completed Forms”.

For an activity  $k$  with  $k \in 1..n-1$ , the corresponding event-response relation might look as follows

$R_k$ : write object po121 in “Role/user<sub>k</sub> work items”  $\Rightarrow$   
    assign(crt\_object, “Role/user<sub>k+1</sub> work items”);  
    delete assign(crt\_object, “Role/user<sub>k</sub> work items”);  
    create deny(crt\_user, {w}, crt\_obj).

The last administrative command prevents a user from signing twice the purchase order (actually, the user could sign the order but not save it back). Finally, for activity  $n$  the event-response relation might look as follows:

$R_n$ : write object po121 in “Role/user<sub>n</sub> work items”  $\Rightarrow$   
    assign(crt\_object, “Approved Orders”);  
    delete assign(crt\_object, “Role/user<sub>n</sub> work items”);  
    delete event/response( $R_0, \dots, R_n$ ).

which sends the purchase order to the “Approved orders” container, from where the Acquisition role can read it. The last command in this relation also deletes all event-response relations related to this purchase order object. For our example, the event-response relations are:

$R_0$ : write object po121 in “Completed Forms”  $\Rightarrow$   
    assign crt\_object to “alice work items”;  
    delete assignment of crt\_object to “Completed Forms”.

R<sub>1</sub>: write object po121 in “alice work items” ⇒  
 assign(cert\_object, “Contracting work items”);  
 delete assign(cert\_object, “alice work items”);  
 create deny(cert\_user, {w}, cert\_obj).

R<sub>2</sub>: write object po121 in “Contracting work items” ⇒  
 assign(cert\_object, “Accounts Payable work items”);  
 delete assign(cert\_object, “Contracting items”);  
 create deny(cert\_user, {w}, cert\_obj).

R<sub>3</sub>: write object po121 in “Accounts Payable items” ⇒  
 assign(cert\_object, “Approved Orders”);  
 delete assign(cert\_object, “Accounts Payable work items”);  
 delete event/response(R<sub>0</sub>, R<sub>1</sub>, R<sub>2</sub>, R<sub>3</sub>).

As noted before, when alice performs activity 1, right after she saves the signed purchase order in her work items container, the PM generates a deny(alice, {w}, po121), according to the event-response relation R<sub>1</sub>. If alice tries to sign the purchase order again as a member of the Accounts Payable role in Activity 3, she would be prevented from saving the purchase order by the above deny. Only Charlie would be able to sign po121 for the Accounts Payable role.

## REFERENCES

- [1] DOD Trusted Computer System Evaluation Criteria, DOD STD 5200.28.
- [2] D. Bell and La Padula. Secure computer systems: unified exposition and MULTICS. Report ESD-TR-75-306, The MITRE Corporation, Bedford, Massachusetts, March 1976.
- [3] D. Brewer and M. Nash. The Chinese wall security policy. In *proc. of the Symp. on Security and Privacy*, pp. 215-228. IEEE Press, 1989.
- [4] David F. Ferraiolo, Serban I. Gavrila, Vincent C. Hu, D. Richard Kuhn: Composing and combining policies under the policy machine. SACMAT 2005: pp. 11-20
- [5] D. Ferraiolo and R. Kuhn. Role-Based Access Control. In *Proc. of the NIST-NSA Nat. (USA) Comp. Security Conf.*, pp. 554-563, 1992.
- [6] D. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, R. Chandramouli, “Proposed NIST standard for Role-Based Access Control”, in *ACM Transactions on Information and Systems Security*, Vol. 4, No. 3, Aug. 2001, pp. 224-274.
- [7] R. Graubart, “On the Need for a Third Form of Access Control,” *Proc. Of the 12<sup>th</sup> National Computer Security Conference*, pp. 296-304 (Oct. 1989).

- [8] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63, 9 (September 1975), 1278–1308.
- [9] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role Based Access Control Models, *IEEE Computer*, 29, 2 (Feb. 1996), 38-47.
- [10] R. Simon and M. Zurko. Separation of duty in role based access control environments. In *Proc. of, New Security Paradigms Workshop*, September 1997.